

nomodo - Manage your system by yourself

Autori (in ordine alfabetico): Giuseppe Glorioso Lucia Polizzi

May 5, 2018

Contents

1 Introduzione

1.1 Introduzione al progetto

Il progetto nomodo nasce dalla necessità di un applicativo di gestione dei sistemi Ubuntu che sia più immediato ed accessibile rispetto al classico terminale, e quindi utilizzabile anche dagli utenti che per un motivo o per un altro non possono o non vogliono avere a che fare con il terminale. Nomodo si prende in carico di eseguire tutte le chiamate al terminale o meno per eseguire operazioni atte alla gestione del sistema presentando all'utente una interfaccia web chiara e comprensibile. Per operazioni in questo caso si intendono l'aggiornamento, la manutenzione e il miglioramento del sistema come ad esempio l'installazione dei pacchetti, la ricerca e la modifica dei file, così come operazioni di più alto livello come la gestione basilare del web server Apache.

1.2 Informazioni tecniche

Python + Flask L'applicativo scritto in python è basato sul framework Flask, utilizzato tra l'altro come webserver per l'accesso al pannello. Durante la fase di sviluppo si è utilizzato nginx come reverse proxy in modo da poter raggiungere il pannello web sulla porta 80 e non sulla 5000. È stata presa poi in seguito la decisione di lasciare che l'applicativo girasse sulla porta 5000 in quanto meno comune e quindi meno alla mercé degli hacker.

L'applicazione è stata quindi divisa in modo netto nelle due componenti fondamentali, il **??** e il **??** che anche andremo quindi ad analizzare qui brevemente e più approfonditamente nei capitoli successivi:

- Il **backend** consiste in una serie di funzioni raccolte in una serie di file a mò di libreria, risiedenti nella cartella `systemcalls` (come ad es. `system.py` o `user.py`), utilizzati sia per la raccolta di dati sia per eseguire azioni sul sistema che non necessitano di output in uscita
- Il **frontend** rappresenta la parte grafica dell'applicativo web, e utilizza le funzioni del backend per la ricerca di informazioni e per la modifica alle componenti del sistema inclusa la modifica dei file quali i file di configurazioni

MongoDB Ogni operazione sensibile effettuata tramite l'applicazione comporta la memorizzazione delle modifiche che comporta la stessa in documento di mongodb, così da poter risalire alla storia delle operazioni effettuate e tentare un revert delle modifiche in caso ad esempio il sistema perda di stabilità o le modifiche non portino al risultato sperato. Tali operazioni possono riguardare ad esempio la modifica di un file, o la rimozione di un pacchetto dal sistema. Ogni log in mongodb presenta inoltre un flag **status** che indica se l'operazione eseguita sia andata a buon fine o meno, in modo da rendere più chiara la navigazione tra i log e dare la possibilità all'utente di filtrarli in base a questo campo. ¹

2 Backend

Come anticipato in sezione **??** il backend è composto da una serie di funzioni raggruppate per categoria che fanno utilizzo di varie librerie python per compiere operazioni che possono o meno alterare lo stato del sistema. Allo stato attuale le categorie che compongono il backend sono le seguenti:

- Utenti

¹ Le uniche operazioni memorizzate nel database sono quelle relative all'utilizzo dell'applicativo; una modifica effettuata direttamente sul sistema ad esempio tramite il terminale va incontro alle regole del sistema Ubuntu e ogni modifica potrebbe essere irreversibile. In questi casi fare riferimento ai log del sistema che è possibile trovare al percorso `/var/log/` o sul pannello web alla sezione Log.

- Network
- Cron
- Sistema
- Apache
- Database
- File
- Logs

Interfaccia al frontend Ogni funzione chiamata restituisce sempre un dizionario contenente almeno n codice di ritorno e il logid del documento inserito in mongo con un mongo `_id` se applicabile² oppure un logid `None` se non è stato creato alcun log. Distinguiamo quindi 2 casi in base al valore della variabile `returncode`:

- Se `returncode = 0` l'operazione è andata a buon fine e il dizionario conterrà una terza variabile `data` che conterrà i dati richiesti se la funzione chiamata è tesa per restituire output oppure sarà una variabile nulla se la funzione non restituisce output
- Se `returncode \neq 0` c'è stato un errore durante l'esecuzione dell'applicazione e il dizionario restituito conterrà quindi una terza variabile `stderr` il cui valore è un messaggio di errore e, se l'errore è dato da un comando eseguito in bash, il comando che una volta lanciato ha generato l'eccezione.

Il frontend o l'utente che voglia chiamare per qualsivoglia motivo le funzioni del backend direttamente, potrà farlo quindi nel seguente modo::

```
data = getifacestat()
if data['returncode'] is 0:
    data = data['data']
else:
    print( data['stderr'] )

pprint(data)
```

subprocess Le funzionalità di Python più utilizzata per la realizzazione dell'applicazione sono senza dubbio quelle appartenenti alla libreria `subprocess`, che permette di eseguire comandi come se si stessero eseguendo in bash. Si è cercato il più possibile di limitare l'utilizzo di questa libreria ma le sue funzionalità si sono rese necessarie nella maggior parte dei casi delle funzioni del backend, a causa della scarsa agilità che ha python di interfacciarsi col sistema sottostante. In generale l'esecuzione di un comando avviene nel seguente modo:

```
command = [ 'ifconfig', '-a' ]
try:
    output = check_output(command, stderr=PIPE, universal_newlines=
        True)
```

² Cioè in caso l'operazione sia una operazione sensibile e richieda quindi un inserimento in mongo per tenere traccia della stessa

```

except CalledProcessError as e:
    return command_error(e, command, logid)

return command_success( output )

```

Tutte le funzioni di nomodo ritornano o con un `??` in caso l'operazione sia andata a buon fine o con un `??` in caso il comando non vada a buon fine e venga lanciata l'eccezione `CalledProcessError`. In entrambi i casi viene restituito il dizionario menzionato in sezione `??`. Un esempio di comando che non restituisce output è il seguente:

```

def removeuser( user , removehome=None ):

    logid = mongolog( locals() , getuser( user ) )

    try:
        command = [ 'deluser' , user ]
        if removehome: command.append( '--remove-home' )

        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error(e, command, logid)

    return command_success(logid)

```

Nelle prossime sezioni verranno analizzate tutte le categorie e spiegato il funzionamento di ogni funzione che contengono.

2.1 utilities

Questa categoria contiene per la maggior parte funzioni che non vengono mai richiamate direttamente dal frontend, ma vengono utilizzate dalle altre funzioni del backend. Fa eccezione la funzione `??` che oltre essere chiamata da queste funzioni può anche essere chiamata direttamente, ed è utile alla modifica dei.

Analizziamo le funzioni di questa libreria nelle prossime sezioni.

2.1.1 mongolog()

```

def mongolog( params , *args ):

    dblog = dict( {
        'date': datetime.datetime.utcnow() ,      #Operation date
        'funname': inspect.stack() [1][3] ,      #Function name
        'parameters': params ,                   #Called function's
            parameters
    } )

    for arg in args:
        dblog.update( arg )

```

```
#ObjectID in mongodb
return db.log.insert_one( dblog ).inserted_id
```

Viene chiamata ogni volta che una funzione sia classificata come **sensibile** cioè che va a modificare lievemente o pesantemente il sistema e prende in carico di creare un log mongodb contenente le operazioni eseguite e i dati modificati dalla funzione. Accetta N parametri di cui il primo (obbligatorio) è la lista di parametri con cui è stata lanciata la funzione di cui si sta memorizzando il log. Ad esempio in

```
def ifacedown( iface ):
    logid = mongolog( locals() )
    ...
```

il primo parametro è **locals()** che contiene la variabile **iface** che verrà quindi memorizzata nel log di mongo; il secondo parametro (opzionale) può essere uno o più dizionari da unire al dizionario memorizzato in mongodb. Ad esempio nella funzione **addusertogroups**:

```
def addusertogroups(user , *groups):

    #Logging operation to mongo first
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals() , userinfo )
    ...
```

Si è deciso che prima di aggiungere un utente a dei nuovi gruppi si va a memorizzare in mongo non solo **locals()** e quindi **user** e ***groups** ma anche le informazioni sull'utente ricavate attraverso la funzione **getuser()** e passate a **mongolog()** come secondo parametro.

Il dizionario di base memorizzato in mongo è formato da tre elementi:

- La data in cui viene effettuata l'operazione
- Il nome della funzione che ha chiamato **mongolog**, ricavata tramite il supporto della libreria **inspect**
- I parametri della funzione che chiama, come spiegato in precedenza, e ottenuti chiamando la funzione **locals()**

2.1.2 mongologstatus() e funzioni collegate

```
def mongologstatus(logid , status):

    return db.log.update_one(
        { '_id' : logid },
        { '$set' : { 'status' : status } },
        upsert=False
```

```

    )

def mongologstatuserr(logid , status='error'):
    return mongologstatus(logid , status)
def mongologstatussuc(logid , status='success'):
    return mongologstatus(logid , status)

```

Parametri Accetta 2 parametri:

- **logid**: è il logid del documento di mongo a cui aggiungere o modificare il campo **status**
- **status='error'**: è lo stato da assegnare la log individuato da **logid**

Questa funzione è intesa per aggiungere o modificare il campo **status** di un log di MongoDB. Le funzioni di nomodo (come è giusto che sia) creano un documento di mongo per memorizzare le informazioni sull'operazione prima di procedere all'operazione stessa. In caso un'operazione non andasse nel modo aspettato bisognerebbe quindi marcare il documento appena creato in mongo in modo da avvisare l'utente che sta consultando il log che l'operazione riferita a quel documento non è andata a buon fine. La memorizzazione del log avviene quindi nei seguenti step:

1. Viene lanciata la funzione che richiede la memorizzazione del log e quindi **??**, che va a creare il log senza nessuna indicazione sul successo o meno dell'operazione
2. Dopo l'esecuzione della funzione viene chiamata **command_success** se l'operazione è andata a buon fine; la prima operazione che questa va ad eseguire è chiamare a sua volta la funzione **mongologstatussuc()** che chiama **mongologstatus()** con il parametro **status='error'** aggiungendo tale campo **status** al log di mongo ed indicando la buona riuscita dell'applicazione all'utente che andrà ad analizzare i log
3. In caso invece la funzione vada in errore viene chiamata **??** che chiama **mongologstatuserr()** che chiama **mongologstatus()** con il secondo parametro **status='error'** aggiungendo tale campo al log di mongo
4. In caso invece si voglia personalizzare il campo **status** basta quindi che la funzione chiami direttamente **mongologstatus()** con il secondo parametro **status** ad un qualsivoglia valore si voglia inserire, ad es. **status='canceled'**

Si intuisce quindi da questi step che un log che non abbia il campo **status** indica un crash della funzione nel codice che è intercorso tra la memorizzazione del log e l'aggiunta del campo **status**.

L'operazione deve fallire se il documento indicato da **logid** non esiste, quindi si è aggiunta la direttiva **upsert=False**.

Ecco un esempio che mostra lo stato di un log appena aggiunto (senza il campo **status**) e al termine dopo aver chiamato **command_success**:

```

> db.log.find()
{ "_id" : ObjectId("5ae596d4bf3bd205c1aeaa25"), "parameters" : { "shell"
  : "/bin/bash", "user" : "giuseppe2", "password" : "test" }, "funname"
  : "adduser", "date" : ISODate("2018-04-29T09:56:36.627Z") }
> db.log.find()

```

```
{ "_id" : ObjectId("5ae596d4bf3bd205c1aeaa25"), "parameters" : { "shell"
  : "/bin/bash", "user" : "giuseppe2", "password" : "test" }, "funname"
  : "adduser", "date" : ISODate("2018-04-29T09:56:36.627Z"), "status" :
  "success" }
```

Return Restituisce un oggetto della classe

2.1.3 command_success

```
def command_success( data=None, logid=None, returncode=0 ):

    if logid:
        mongologstatussuc( logid )

    return dict({
        'returncode': returncode,
        'data': data,
        'logid': logid
    })
```

La funzione **command_success** fondamentalemente costruisce il dizionario da restituire all'utente quando una funzione del backend ha finito le sue operazioni e non ci sono stati errori durante l'esecuzione. Insieme alla sorella ?? sono le uniche due funzioni chiamate al termine di una funzione del backend.

Parametri Accetta tre parametri:

- **data=None**: Sono i dati da restituire all'utente se la funzione che l'ha chiamata li genera. Di base è **None**
- **logid**: Il logid a cui aggiungere il campo **status** e da restituire all'utente nel dizionario come campo del dizionario. Di base è **None** in quanto il chiamante potrebbe non aver generato un mongolog per l'operazione che ha effettuato
- **returncode**: È il codice di ritorno che verrà inserito nel dizionario. Essendo questa funzione invocata ogni qualvolta il chiamante esegue tutte le operazioni senza errore di base questo parametro è 0 ad indicare successo e può quindi essere omesso, ma può essere personalizzato passandolo alla chiamata

Funzionamento La prima operazione eseguita è la chiamata **amongologstatussuc()** per aggiungere al log di mongo il campo **status**, questo solo in caso il parametro **logid** sia non nullo e quindi la funzione chiamante ha dovuto memorizzare un mongolog. Successivamente va a costruire il dizionario da restituire formato dal codice di ritorno, i dati voluti dall'utente (se disponibili, altrimenti **None**), e il *logid* dell'operazione.

Return Restituisce il dizionario contenente i parametri passati alla funzione o i loro valori di default se non vengono passati. Da utilizzare come spiegato in sezione ??.

2.1.4 command_error

```
def command_error( e=None, command=None, logid=None, returncode=1, stderr
                  ='No messages defined for this error' ):

    if logid:
        mongologstatuserr( logid )

    return dict({
        'returncode': e.returncode if e else returncode,
        'command': ' '.join(command),
        'stderr': e.stderr if e else stderr,
        'logid': logid
    })
```

`command_error` è l'opposto di `command_success`. Come visto in sezione ?? viene invocato quando un comando lanciato attraverso la libreria `subprocess` fallisce nell'esecuzione. Può essere però anche usato per generare un dizionario di errore personalizzato.

Parametri Accetta 5 parametri:

- **e=None**: è l'oggetto creato nel caso in cui venga lanciata l'eccezione `CalledProcessError`
- **command=None**: è il comando la cui esecuzione ha generato l'eccezione
- **logid**: Il logid a cui aggiungere il campo `status` e da restituire all'utente nel dizionario come campo del dizionario. Di base è `None` in quanto il chiamante potrebbe non aver generato un mongolog per l'operazione che ha effettuato
- **returncode=1**: Un codice di ritorno personalizzato da inserire nel dizionario in caso il parametro `e` sia nullo
- **stderr='No messages defined for this error'**: È il messaggio di errore da inserire nel dizionario in caso il parametro `e` sia nullo

Funzionamento La funzione costruisce un dizionario da restituire all'utente contenente le varie informazioni sull'errore che è accaduto, ossia il codice di ritorno, il messaggio di errore, il comando che ha generato l'errore (da passare in input) e il logid del documento in mongo che riguarda il comando/operazione.

Distinguiamo 3 casi:

- Viene generato un oggetto del tipo `CalledProcessError` appartenente a `subprocess`: in questo caso si passa alla funzione l'oggetto generato e il comando che ha causato l'errore. La funzione ricava automaticamente da questo oggetto il codice e il messaggio di errore e lo inserisce nel dizionario insieme al comando di `subprocess` che ha causato l'errore. È quindi in questo caso necessario passare almeno questo oggetto e il comando (che non è però strettamente necessario)
- Si vuole generare un errore personalizzato: in questo caso invece i parametri `e` e `command` non devono essere passati, e al loro posto vengono passati `returncode` e `stderr` che verranno inseriti nel dizionario da restituire.

- Si vuole generare un errore di default: in quest'ultimo caso basta chiamare la funzione senza passare alcun parametro e viene generato un errore di base in cui i valori del codice di ritorno saranno quelli assegnati di default e che si può vedere nella sezione *Parameters*

Oltre a restituire il dizionario di errore questa funzione, se il parametro `logid` è non nullo agisce come ?? aggiungendo quindi al log dell'operazione il campo `status` col valore `error`.

Return Restituisce il dizionario creato come descritto nel funzionamento e come si può vedere nel codice.

2.1.5 filedit

```
def filedit(filename, towrite=None, force=False):

    if not towrite:
        with open(filename, 'r') as opened:
            return opened.read()

    if not force:
        #If force is not specified then calculate md5sum to check whether
        the file has changed.
        #If file hasn't changed it is not written
        md5new = hashlib.md5()

        ##(Referring encode()) To calculate md5sum string 'towrite' needs
        to be converted into 'bite' format
        md5new.update( towrite.encode() )
        md5new = md5new.hexdigest()

        #old file content md5sum
        md5old = hashlib.md5( open( filename, 'rb' ).read() ).hexdigest()

        if md5new == md5old:
            return command_error( returncode=2, stderr='Nothing to write(
                no changes from original file) . You can force writing using
                the parameter "force=True" ' )

    ##This code will get executed on either Force==True or md5new !=
    md5old

    #Better insert the diff between the 2 files instead of full content,
    thus
    #we need to remove the parameter "towrite" from "locals()" and pass
    the dict() returned by the filediff() function to mongolog()
    localsvar = locals()
    del localsvar['towrite']
    logid = mongolog( localsvar, filediff(filename, towrite) )
```

```
#Writing new content to "filename" file
opened = open(filename , 'w')
opened.write(towrite)
opened.close()

return command_success( logid=logid )
```

Questa funzione è intesa per dare un supporto in caso l'utente voglia modificare un file. Implementa tutte le operazioni e i controlli che bisognerebbe effettuare prima della modifica di un file, incluso un mongolog da cui si possa risalire al contenuto del file prima della scrittura. È inoltre capace di restituire il contenuto di un file, utile in fase di pre-scrittura.

Parametri Sono 3 i paramentri accettati da questa funzione:

- **filename**: è il percorso del file sul sistema che si vuole modificare
- **towrite=None**: è il nuovo contenuto del file da scrivere sullo stesso
- **force=False**: questa variabile (di base a **False**) se **True** forza la scrittura del nuovo contenuto anche se questo non differisce dal contenuto originale del file

Funzionamento

```
if not towrite:
    with open(filename , 'r') as opened:
        return opened.read()
```

Oltre a scrivere i file questa funzione è intesa anche per ritornare il contenuto di un file. Come si vede dal primo **if** se **towrite** è nulla semplicemente apre il file in lettura e restituisce il suo contenuto.

```
if not force:
    md5new = hashlib.md5()

    md5new.update( towrite.encode() )
    md5new = md5new.hexdigest()

    md5old = hashlib.md5( open( filename , 'rb' ).read() ).hexdigest()

    if md5new == md5old:
        return command_error( returncode=2, stderr='Nothing_to_write(
            no_changes_from_original_file).You_can_force_writing_
            using_the_parameter_"force=True"' )
```

In questa seconda parte controlla la varibile **force**. Se questa è **false** genera l'md5 del nuovo contenuto (**towrite**) e del vecchio contenuto (quello del file) e nell'ultimo **if** ne controlla l'uguaglianza. Quindi se **force=False** e **md5new==md5old** non c'è necessità di scrivere il file e genera quindi un messaggio di errore personalizzato (o potremmo dire di warning in questo caso) usando la funzione **??**, e che avverte il chiamante (o il frontend) che non è stata eseguita alcuna operazione ma che la si può forzare passando il parametro **force** col valore booleano **True**.

```
localsvar = locals()
del localsvar['towrite']
logid = mongolog( localsvar , filediff(filename , towrite) )

#Writing new content to "filename" file
opened = open(filename , 'w')
opened.write(towrite)
opened.close()

return command_success( logid=logid )
```

Si arriva quindi a questo pezzo di codice se `force=True` o se `force=False` e `md5new!=md5old`. Questa funzione andando a modificare file che possono essere o meno importanti necessita di un log che memorizzi una quantità di informazioni tali da consentire il ripristino del file vecchio. Si intuisce però che memorizzare interamente il vecchio e il nuovo contenuto sarebbe impensabile per avere un mongolog decente e di piccole dimensioni. Si è quindi optato per inserire il solo diff tra vecchio e nuovo contenuto. In casi comuni e per i nostri scopi questa scelta porta ad una riduzione significativa della dimensione del log, in quanto raramente un utente cancella completamente e riscrive file di migliaia di righe di codice.

Passando al funzionamento quindi si memorizza dapprima ciò che restituisce la funzione `locals()` (che come spiegato restituisce i parametri con cui la funzione è stata chiamata) e si inserisce ciò che ritorna nella variabile `localsvar`. Dopodichè da questa variabile (che è diventata un dizionario dopo l'assegnazione) si elimina il parametro `towrite` che a quanto ne sappiamo potrebbe anche essere di migliaia di righe. Si crea a questo punto un mongolog che non abbia più al suo interno il contenuto scritto ma ciò che ritorna la funzione `??` che come vedremo in seguito non fa altro che restituire il diff tra il vecchio e il nuovo contenuto del file. Una volta effettuato i controlli e memorizzati in mongo i dati necessari si procede finalmente a scrivere il nuovo contenuto sul file tramite le funzioni Python per la gestione dei file. Si invoca infine `command_success` passandogli il `logid`.

Return Restituisce il *dizionario di successo* generato dalla funzione `command_success` con all'interno il solo `logid` dell'operazione.

2.1.6 filediff

2.2 Utenti

2.3 Network

2.4 Cron

2.5 Sistema

2.6 apache

2.7 Database

2.8 File

2.9 Logs

3 Frontend

Frontend

4 Utility

4.1 Red Hat Developer Toolset

4.2 kerberos5

1. Copiare il file `/etc/krb5.conf` da uno dei nodi di cresco 4
2. Abilitare kerberos come tipologia di autenticazione:
`authconfig --enablekrb5 --updateall`
3. Utilizzare `klog.krb5` come `klog` di default:
 - `mv /usr/bin/klog /usr/bin/klog.orig`
 - `ln -s /usr/bin/klog.krb5 /usr/bin/klog`
4. Copiare il keytab da `cresco-inst1`:
`cp /afs/enea.it/system/arc/keytab/services/host/cresco4x002.portici.enea.it /etc/krb5.keyta`
5. Link simbolici alle shell? Discuterne con Guido
6. Creare un link simbolico per `pagsh`:
`ln -s /usr/bin/pagsh /usr/afsws/bin/pagsh`

4.3 Rimossi tra CentOS 6 e 7 e le cui alternative non presenti su CRESCO 6

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 non è presente nei sistemi di CRESCO 6, e che si dovrebbe quindi provvedere ad installare:

Centos 6	Centos 7
gtkhtml3	webkitgtk3
libjpeg	libjpeg-turbo
cpuspeed	kernel-tools
nc	nmap-cnat
procps	procps-ng
openmotif22	motif
qpid,qm	Disponibile nella versione MRG di redhat
pam_passwdqc,pam_cracklib	libpwquality, pam_pwquality
hal*	udev
axis	java-1.7.0-openjdk
classpath[x]?-jaf	java-1.7.0-openjdk
classpath[x]?-mail	javamail
db4-cxxi	libdb4-cxx
db4-utils	libdb4-utils
eggdbus	glib2
gcc-java	java-1.7.0-openjdk-devel
GConf2-gtk	GConf2
geronimo-specs	geronimo-parent-poms
geronimo-specs-compatible	geronimo-jms, geronimo-jta
hal-devel	systemd-devel
ibus-gtk	ibus-gtk2
jakarta-commons-net	apache-commons-net
junit4	junit
m17n-contrib-*	m17n-contrib
m17n-db-*	m17n-db,m17n-db-extras
seekwatcher	iowatcher
udisks	udisks2
unique	unique2,glib2
unix2dos	dos2unix

4.4 Rimossi da Centos 6 e 7 e le cui alternative sono presenti su CRESCO 6

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 è presente nei sistemi di CRESCO 6, e che quindi non è necessario installare:

Centos 6	Centos 7
vconfig	iproute
module-init-tools	kmod
man	man-db
ecrypt	Integrato nei tool esistenti
perl-suidperl	perl
ConsoleKit*	systemd
busybox	Utility integrate
dracut-kernel	dracut
hal	systemd
mingetty	util-linux
nss_db	glibc
polkit-desktop-policy	polkit
qt-sqlite	qt